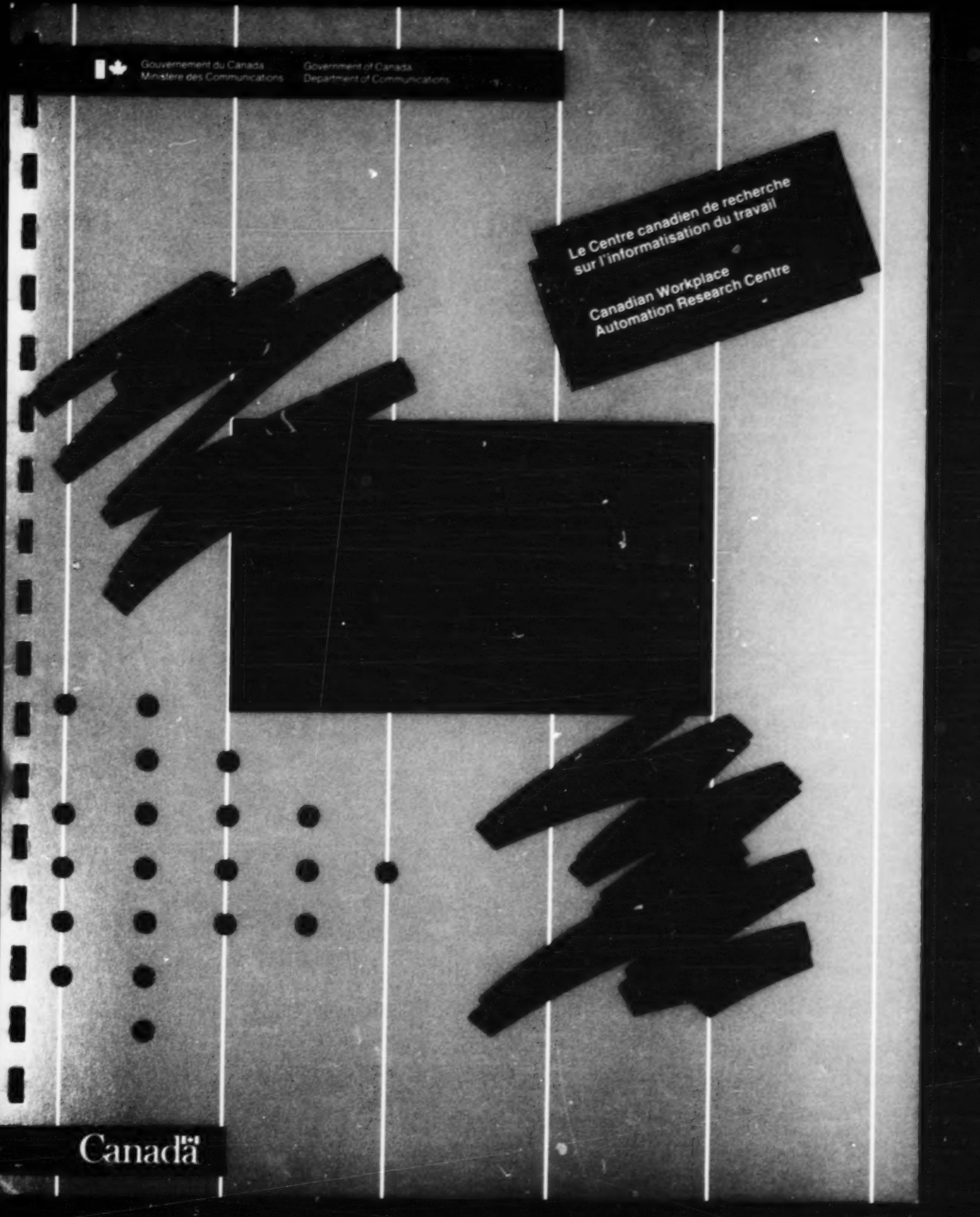

Gouvernement du Canada
Ministère des Communications
Government of Canada
Department of Communications
Le Centre canadien de recherche sur l'informatisation du travail
Canadian Workplace Automation Research Centre
Canada

RATP: A NEW FORMALISM IN THE UNIFICATION GRAMMAR FAMILY

Marc Dymetman

Canadian Workplace Automation Research Centre

Department of Communications Canada

Laval

April 87

N° de cat.
ISBN

The views expressed in this report are those of the author only.

* Ce rapport est disponible aussi en français.

1 - INTRODUCTION

1.1 Overview of RATP

RATP is a formalism for writing Unification Grammars that integrates, in a natural way, a component giving it the power of a general programming language. This "programming language" component has a structure analogous to a set of Horn clauses in pure Prolog, with the difference that *DAGs* (Directed Acyclic Graphs) rather than *terms* act as basic data structures.

The inclusion of such a component adds to the modularity of the grammar formalism, thus allowing easy definition and reuse of groups of grammatical features. It also helps clarify the concepts of conjunction, disjunction and negation as applied to functional structures and provides for introducing the possibility of delayed evaluation ("goal-freezing").

A compiler has been written that takes an RATP grammar and produces Prolog Code. This translation is made easier by a simple encoding of DAGs as terms and by the direct application of Prolog's unification mechanism to the result of this encoding.

This embedding of RATP in Prolog has the advantage that it provides a powerful programming environment (in this case MProlog) and that, in view of the parallelism that exists between the structure of an RATP program and a Prolog one, the two languages can easily be made to communicate in a natural way.

1.2 Unification Grammars

What is referred to here by the general term "Unification Grammars" actually corresponds to a group of grammar formalisms, of which the best

known are: Lexical Functional Grammar (Bresnan, Kaplan 1983), Unification Grammar (Kay 1983) and PATR II (Shieber 1984).

Though they differ, these formalisms share certain important characteristics:

- Grammatical structures are seen as recursive structures which associate values with labels, where these values in turn may be structures of the same type.

- These structures, which go by a variety of names ("feature structures", "functional structures", "f-structures", etc), may be seen from a formal standpoint as special cases of the DAG (for "Directed Acyclic Graph") concept.

- In these structures, it is possible to access the values of various levels by following "access paths" consisting of chains of labels.

- Substructures within the structure may be "shared", that is a single substructure may be accessed through different access paths starting from the same structure above it.

- The basic mechanism for computing grammatical structures is a composition process which "takes" several relatively elementary structures and "assembles" them into a more complex structure, using a *unification* mechanism.

1.3 Parallels between Unification Grammars and Logical Grammars. DAGs and Terms

Another approach currently in vogue in the field of grammar formalisms goes by the generic name of "logical grammars". This term describes those grammar formalisms built "around" Prolog. As is well known, this language was specifically designed for the purpose of grammatical

description. It has given rise to a variety of formalisms. Particularly noteworthy are: "metamorphosis grammars" (Colmerauer 1978), "DCG" (Pereira, Warren 1980), "extraposition grammars" (Pereira 1981) and "gapping grammars" (Dahl 1985).

Logical grammars have a great deal in common with unification grammars. In both cases, the mechanism for unifying grammatical structures plays a fundamental role. In fact, the difference between the two classes of formalism lies essentially in the type of basic data structure manipulated by the systems: *DAGs* in the case of unification grammars, and *terms* in the case of logical grammars.

While in a DAG substructures are accessed by specifying a label (or string of labels) and may be of indefinite length, in a term substructures are differentiated by their positions as arguments, the number of arguments in a term being fixed once and for all. Furthermore, the notion of accessing the "deep" substructures of a structure is much less relied on in the case of terms than in the case of DAGs; the ease of describing such access in unification grammars leads to a different programming style which seems to have the advantage of clarity for writing grammars.

Starting from this fact that the two kinds of formalisms are distinguished essentially by their basic data structures, it is natural to ask:

(1) Whether DAGs may be encoded as terms and whether terms may be encoded as DAGs.

(2) Whether the unification mechanism for terms and the unification mechanism for DAGs may not be reduced to a common underlying mechanism.

The answer to both questions is yes, as the rest of the report shows in greater detail. Given that these two properties are real, it is possible

to use the Prolog unification mechanism to effectively implement unification grammars, more specifically a new formalism for the class of unification grammars: RATP.

1.4 RATP as a grammar formalism and as a programming language

RATP is a grammar writing language which, within the class of "unification grammars", resembles the PATR formalism (Shieber 1984), but which introduces certain innovations (general predicates on structures).

An RATP grammar is made up of two types of textual constructions:

- augmented context-free grammar rules;
- predicate definitions, in the form of a particular type of Horn clauses.

- Each grammar rule is composed of:

  - a context-free "skeleton"
  - the "flesh" on the bones of this skeleton(!), including a sequence of equations and predicates which have to be verified by the grammatical structures associated with the skeleton's non-terminals and terminals, or by the substructures within these structures. Such substructures are accessed by access paths formed of labels bearing some grammatical meaning.

- Predicate definitions are stated in the form of Horn clauses, as in Prolog. The essential difference from Prolog is that the objects to which the predicates apply are "feature structures" (DAGs), whereas in Prolog they are terms.

The introduction of predicate definitions of this type considerably enhances the power of the formalism, as these predicates make it possible to specify arbitrarily complex classes of DAGs and use them to encapsulate sets of structural constraints.

In fact, the introduction of these predicates gives the formalism the power of a universal programming language similar to Prolog (see the "multiplication" example in section 2), which is of secondary importance here, yet with the significant consequences that new light is shed on the concepts of conjunction, disjunction and negation, and that some useful Prolog concepts (eg "freeze") can be naturally imported into the formalism.

Another, and perhaps the most important consequence of this approach is that the formalism highlights the fact that "unification grammars" and "logical grammars" are very close variants of a common paradigm, a fact which certain schools of thought tend to obscure.

2 RATP FORMALISM

We shall begin by giving a few examples of structures produced by RATP analysis[1], on which we will comment briefly; we will then describe the structure of the grammar which made these results possible.

2.1 Examples of structures produced by RATP

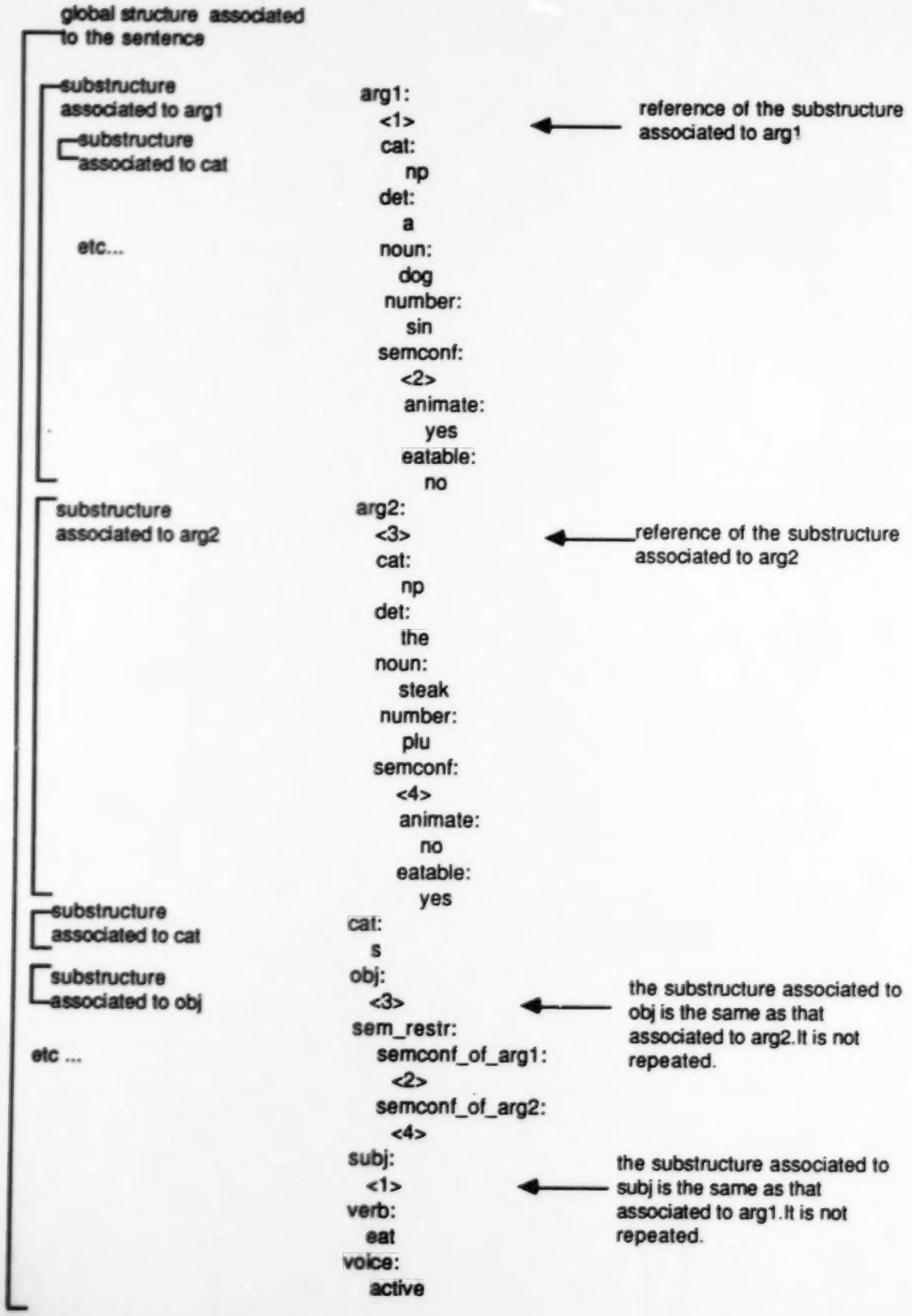
2.1.1 Examples

Example 1: Analysis of "a dog eats the steaks"

[a, dog, eats, the, steaks]

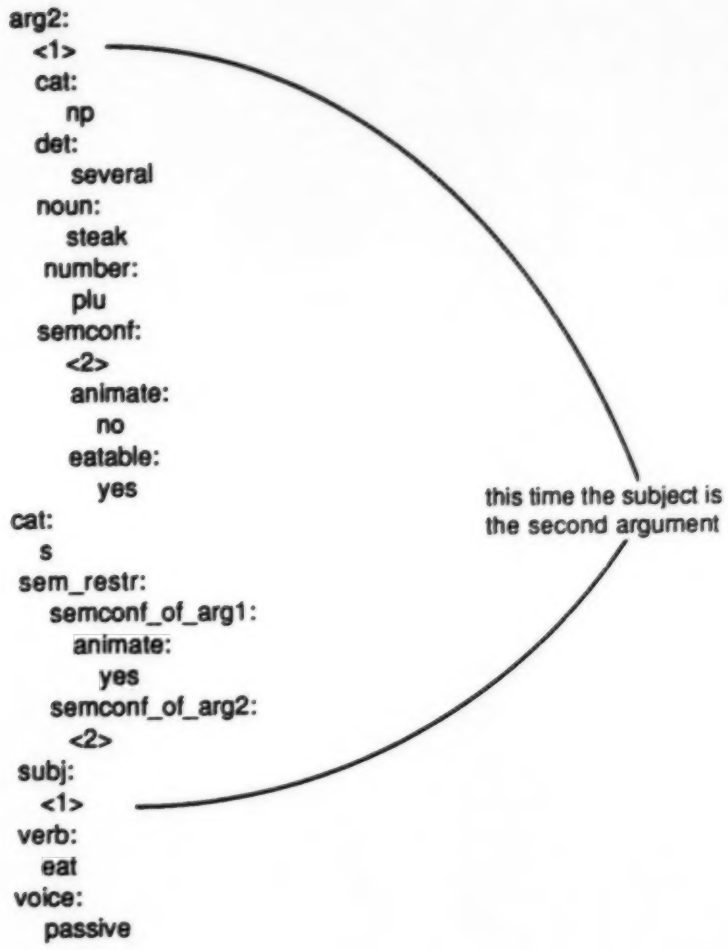
---

[1] It is worth mentioning that RATP may be used for both synthesis and analysis, a feature which this formalism shares with DCG. In both RATP and DCG, the difference between analysis and synthesis is simply a difference in the propagation of instantiations of variables.


global structure associated to the sentence
substructure associated to arg1
substructure associated to cat
etc...
substructure associated to arg2
substructure associated to cat
substructure associated to obj
etc ...
arg1:
<1>
cat:
np
det:
a
noun:
dog
number:
sin
semconf:
<2>
animate:
yes
eatable:
no
arg2:
<3>
cat:
np
det:
the
noun:
steak
number:
plu
semconf:
<4>
animate:
no
eatable:
yes
cat:
s
obj:
<3>
sem_restr:
semconf_of_arg1:
<2>
semconf_of_arg2:
<4>
subj:
<1>
verb:
eat
voice:
active
reference of the substructure associated to arg1
reference of the substructure associated to arg2
the substructure associated to obj is the same as that associated to arg2.It is not repeated.
the substructure associated to subj is the same as that associated to arg1.It is not repeated.

Example 2: Analysis of "several steaks are eaten"

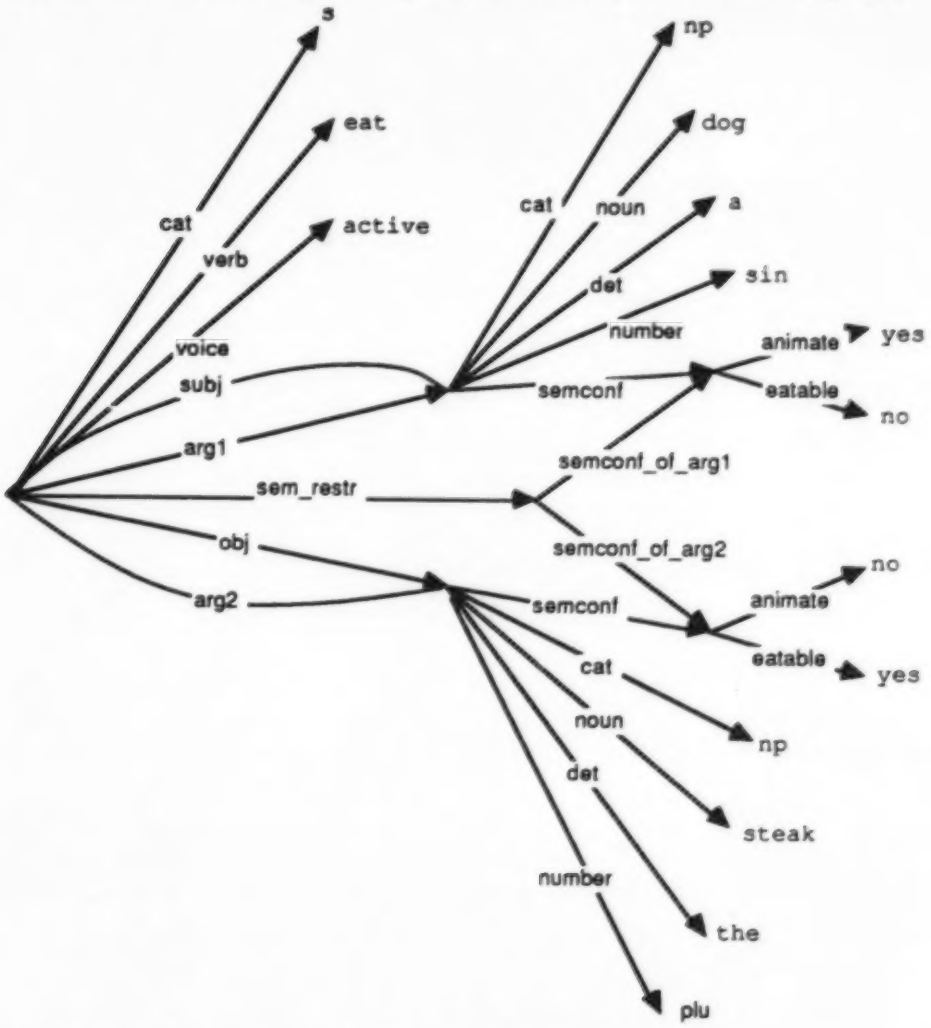
[Several, steaks, are, eaten]



2.1.2 Explanation of examples

2.1.2.1 Notation

Let us examine example 1. The structure given represents the result of the analysis of the sentence: "a dog eats the steaks". This structure is a DAG[2], which can be graphically represented in the following way:



DAG associated with "a dog eats the steaks"

---

[2] Strictly speaking, a distinction should be drawn between **ground DAGs** and **non-ground** (or **variable**) DAGs, but this distinction need not concern us here. The structure shown is in fact a **non-ground** DAG, it represents an infinite family of **ground** DAGs.

Preliminary remarks:

In the resulting structure on page 6, the order of the labels *arg 1, arg 2, cat, obj, subj, verb, voice* is alphabetic. This "vertical" order has no special significance, and a different order is used in the graph on page 8 *(cat, verb,* etc) for reasons of visual clarity.

The DAG represented on page 7 is the structure resulting from analysis of the sentence "a dog eats the steaks". This DAG has the following properties:

- its "cat" (category) has the value "s" ("s" is a degenerate DAG consisting of just one symbol)

- its "verb" has the value "eat" (idem)

- its "voice" has the value "active" (idem)

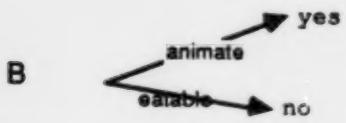
- its "subj" has the value of a DAG (non-degenerate) whose "cat" has the value "np", etc.

- The value of its "arg 1" is the same DAG as for its "subj". In the graph (p 8), this is apparent in that the two DAGs have the same root, while in the "textual" representation (p 6), it is indicated by the fact that the labels "arg 1" and "subj" are associated by the cross-reference (1). Convention dictates that in a DAG -- on paper -- the content of two identical sub-DAGs is not repeated; cross-references of this kind are used instead, thus simplifying the writing, but only in the case of non-degenerate DAGs that appear more than once[3].

---

[3] 3 These cross-references also have an "intensional" value: two cross-referenced entities have been unified at some point and will remain so if the structure is used again subsequently. This fact would not be

The concept of access paths can be illustrated in this example as follows: if the "overall" DAG is designated A, then the access path A.arg1.semconf (where "arg 1" and "semconf" are labels and A is the constant which represents the overall DAG) is an "access path" to DAG B, where B is the constant which represents the DAG:



This same entity B can be accessed through other access paths, namely A.subj.semconf A.sem_restr.semconf_of_arg1 (shared substructures).

2.1.2.2 Linguistic remarks

The grammar which produces these structures (the grammar presented in section 2.2), although quite rudimentary, generates structures which may be read on several levels, in particular "syntactic" and "logical". On a syntactic level, the structure in example 1 ("a dog eats the steaks") produces a "verb", a "subj" and an "obj", while on a logical level, "verb", "arg 1" and "arg 2" appear.

It should be noted that in this example, the substructures "accessed" by the labels "subj" and "arg1" are identical, as are those "accessed" by the labels "obj" and "arg2", which, for transitive verbs, corresponds to the active voice: the syntactic subject and the first logical argument are the same, and the syntactic object and the second logical argument are the same[4].

---

apparent if we were to be content with *printing* the content of the two structures.

[4] There is an obvious analogy with both the "dependency" structures of GETA and the functional structures of Lexical Functional Grammar. In

On the other hand, in example 2 -- "several steaks are eaten" -- there is only one argument present on a logical level, and this is expressed syntactically by the subject function "subj". This matches one of the patterns for the passive voice, the other being the case where arg1 shows up syntactically as the function "by-phrase".

2.2 The RATP grammar that produces these examples

2.2.1 The grammar

By way of example, here is the (very elementary) RATP grammar by which the examples in section 2.1 were analyzed.

A. Augmented context-free rules

```
S -> NP VP,
S.cat=s,
NP.cat=np,
VP.cat=vp,                                    (1)
number_agreement(NP,VP),
S.subj=NP,
verbal_head_features_sharing(S,VP),
voice_treatment(S);

VP -> V,
VP.cat=vp,
V.type=finite,
intransitive_verb(V),                         (2)
VP.voice=active,
verbal_head_features_sharing(VP,V),
VP.number=V.number;

VP -> V NP,
VP.cat=vp,
NP.cat=np,                                    (3)
V.cat=v,
V.type=finite,
transitive_verb(V),
VP.voice=active,
VP.obj=NP,
```

---

these, as also here, the structures associated with arguments are of the same kind as those associated with syntactic functions. In fact, these structures are *shared* by logical and syntactic functions.

```
VP.number=V.number,
verbal_head_features_sharing(VP,V),
VP.sem_restr.semconf_of_arg2=NP.semconf;

VP -> AUX V,
VP.cat=vp,
AUX.cat=aux,
be(AUX),
V.cat=v,                                        (4)
V.type=past_participle,
transitive_verb(V),
VP.voice=passive,
VP.number=AUX.number,
verbal_head_features_sharing(VP,V);

AUX -> is,
be(AUX),                                        (5)
finite_singular(AUX);

AUX -> are,
be(AUX),                                        (6)
finite_plural(AUX);

NP -> Det Noun,
NP.cat=np,
Det.cat=det,
Noun.cat=noun,
NP.det=Det.det,
number_agreement(NP,Det),                       (7)
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;

Noun -> dog,
Noun.noun=dog,
sinnoun(Noun),                                  (8)
high_status_animal(Noun);

Noun -> rats,
Noun.noun=rat,                                  (9)
plunoun(Noun),
high_status_animal(Noun);

Noun -> steaks,
Noun.noun=steak,
plunoun(Noun),                                  (10)
standard_eatable(Noun);

V -> sleep,
intransitive_verb(V),
V.verb=sleep,                                   (11)
animate_arg1(V),
finite_plural(V);

V -> dreams,
intransitive_verb(V),
V.verb=dream,                                   (12)
animate_arg1(V),
```

```
finite_singular(V);

V -> eats,
transitive_verb(V),
animate_arg1(V),
eatable_arg2(V),                                    (13)
V.verb=eat,
finite_singular(V);

V -> eaten,
transitive_verb(V),
animate_arg1(V),
eatable_arg2(V),                                    (14)
V.verb=eat,
past_participle(V);

Det -> a,
Det.cat=det,                                        (15)
Det.det=a,
Det.number=sin;

Det -> the,
Det.cat=det,
Det.det=the,                                        (16)
Det.number=DetNbr,
freeze(DetNbr,sin_or_plu(Det));

Det -> several,
Det.cat=det,                                        (17)
Det.det=several,
Det.number=plu;
```

B. Predicate definitions on DAGs

```
sinnoun(Noun):
Noun.cat=noun,                                      (1)
Noun.number=sin;

plunoun(Noun):
Noun.cat=noun,                                      (2)
Noun.number=plu;

sin_or_plu(X):
X.number=sin;
                                                    (3)
sin_or_plu(X):
X.number=plu;

number_agreement(X,Y):
X.number=Y.number;                                  (4)

finite_plural(Verb):
Verb.type=finite,                                   (5)
Verb.number=plu;

finite_singular(Verb):
```

```
Verb.type=finite,                                    (6)
Verb.number=sin;

past_participle(V):
V.type=past_participle;                              (7)

be(AUX):
AUX.cat=aux,                                         (8)
AUX.verb=be;

transitive_verb(V):
V.cat=v,                                             (9)
V.transit=trans;

intransitive_verb(V):
V.cat=v,                                             (10)
V.transit=int;

verbal_head_features_sharing(X,Y):
X.sem_restr=Y.sem_restr,
X.obj=Y.obj,
X.subj=Y.subj,
X.arg1=Y.arg1,                                       (11)
X.arg2=Y.arg2,
X.verb=Y.verb,
X.voice=Y.voice;

voice_treatment(S):
S.voice=active,
S.arg1=S.subj,
S.arg2=S.obj,
S.sem_restr.semconf_of_arg1=S.arg1.semconf;
                                                     (12)
voice_treatment(S):
S.voice=passive,
S.arg2=S.subj,
S.sem_restr.semconf_of_arg2=S.arg2.semconf;

animate(X):
X.semconf.animate=yes;                               (13)

nonanimate(X):
X.semconf.animate=no;                                (14)

eatable(X):
X.semconf.eatable=yes;                               (15)

noneatable(X):
X.semconf.eatable=no;                                (16)

high_status_animal(X):
animate(X),                                          (17)
noneatable(X);

standard_eatable(X):
eatable(X),                                          (18)
nonanimate(X);
```

```
animate_arg1(V):
V.sem_restr.semconf_of_arg1.animate=yes;  (19)

eatable_arg2(V):
V.sem_restr.semconf_of_arg2.eatable=yes;  (20)
```

2.2.2 Explanation of the grammar

The grammar in the preceding section is very elementary, but it illustrates the most important aspects of the RATP formalism.

We have divided the grammar into two parts:

- Part A consists of a set of "augmented context-free rules"

- Part B consists of a set of "predicate definitions on DAGs"

The only purpose of this division here is to make reading easier. In an RATP grammar, both types of construction can be arbitrarily mixed.

2.2.2.1 Augmented context-free grammar rules

Let us consider Rule (1)

```
S -> NP VP,                              a
S.cat=s,                                 b.1
NP.cat=np,                               b.2
VP.cat=vp,                               b.3
number_agreement(NP,VP),                 b.4
S.subj=NP,                               b.5
verbal_head_features_sharing(S,VP),      b.6
voice_treatment(S);                      b.7
```

In this rule, line (a) and lines (b.1) through (b.7) have different roles:

- line (a) -- the "skeleton" of the rule -- describes a context-free rule. Here, S, NP and VP are non-terminal nodes in the classic sense.

- lines (b.1) through (b.7) -- the "body" of the rule -- describe relations (or constraints) which have to be verified between substructures (that is, between DAGs). Here, S, NP and VP are variables representing DAGs, namely DAGs associated with the non-terminal nodes S, NP and VP in the skeleton.

It must therefore be kept in mind that the symbols S, NP and VP play a double role in a grammar rule: in the skeleton they are non-terminal nodes, and in the body they are variables designating DAGs which represent the structures obtained for each of the non-terminal nodes. It should be noted in passing that in this role of variables designating DAGs (identifier beginning with a capital letter and appearing in the body of a rule), an identifier may appear "freely", ie not associated with a non-terminal node (eg DetNbr in rule (16)); it then plays an auxiliary role in verifying constraints.

Now let us consider one by one lines (b.1) through (b.7) in rule (1).

(b.1) states an equality constraint. It may be read in the following way: the sub-DAG (substructure) accessed from the DAG associated with S via the access path (consisting of a single label) "S.cat" is equal[5] to the DAG (degenerate, that is, reduced to a symbol) "s" (a constant).

In other words, the category of the DAG associated with S is "s".

(b.2) and (b.3) can be understood in the same way.

(b.4) expresses a two-argument "predicative" constraint, namely that the "number-agreement" relation must be verified between the two DAGs associated with NP and VP. To know the specification of this constraint, one has to refer to definition (6) of the "number-agreement" predicate in Part B of the grammar. This predicate reads: number-agreement (X,Y)

---

[5] or rather "unifiable" (see Section III)

is true when X and Y are two DAGs such that the sub-DAG of X accessed by the access path "X.number" is equal to the sub-DAG of Y accessed by the access path "Y.number". In other words, the "number" of X is equal to that of Y.

(b.5) expresses an equality constraint: the sub-DAG of S accessed by the access path S.subj is equal to the DAG associated with NP. In other words, the "subj" (subject) of S is the whole NP (that is, the entire structure associated with NP).

If (b.1) and (b.5) are compared, it can be seen that, from the point of view of the formalism, both a simple grammatical feature like "s" and a complex structure like that associated with a noun phrase are treated in the same way. They are both sub-DAGs of DAGs, one degenerate and the other complex, accessed by the access paths "S.cat" and S.subj" respectively.

(b.6) expresses a predicative constraint (having two arguments), namely that there is "sharing" of "verbal_head_features" (subj, obj, arg1, arg2, verb, voice, sem_restr) between S and VP.

The content of this constraint is spelled out in the definition (definition II in B) of the predicate "verbal_head_features sharing".

(b.7) expresses a predicative constraint (with one argument), "voice treatment", which has to be verified by the DAG S. On referring to the definition of the "voice_treatment" predicate (B, definition (12), we notice a new aspect of the formalism: there are two "statements" corresponding to the "voice_treatment" predicate. These two statements may be read as a disjunction of conditions: for DAG S to verify the voice-treatment constraint, requires (among other things) that the sub-DAG of S accessed by the access path S.arg2 be equal to the sub-DAG of S accessed by the access path S.obj, while in the case of the passive voice, the constraint requires (among other things) that the sub-DAG of S accessed by the access path S.arg2 be equal to the sub-DAG of S

accessed by the access path S.subj. Section 2.1 shows examples of these two basic types.

2.2.2.2 Predicate definitions

The comments in the preceding section have already illustrated the role of predicate definitions; these definitions make it possible to describe the constraints between DAGs independently of any grammar rule, and also:

(1) to reuse the same relation in several different rules (e.g., the "transitive-verb" definition is used in several different grammar rules)

(2) to considerably augment the language's descriptive power (see section 2.3).

There is a formal similarity between the predicate definitions of RATP and those of pure Prolog:

- A predicate definition contains a certain number of statements (two in the case of "voice_treatment")

- In statements, a distinction is made between the "head" (e.g., "voice treatment") and the "body", or "tail" (S.voice=active, S.arg1=S.subj, S.arg2=S.obj, S.sem restr.semconf_of arg1=S.arg1.semconf)

- Identifications of variables begin with a capital letter (e.g., "S")

- Identifications of constants begin with a lower-case letter (e.g., "active")[6]. This convention is analogous to that used in some dialects of Prolog (Edinburgh syntax).

Certain differences should also be noted:

- Variables designate DAGs rather than terms.

- Substructures are accessed by access paths (e.g., "S.sem restr.semconf of arg2"), whereas in Prolog there is no explicit concept of access to subterms but only implied access determined by the position of a subterm below a term.

2.3 RATP as a formalism for writing grammars and as a programming language

2.3.1 RATP as a formalism for writing grammars

As a formalism for writing grammars, RATP is a language which falls exactly midway between PATR (representing the class of unification grammars) and DCGs (representing the class of logical grammars).

Let us consider the following three rules, taken from the grammar in section 2.2.1:

```
NP -> Det Noun,
NP.cat=np,
Det.cat=det,
Noun.cat=noun,
NP.det=Det.det,
number_agreement(NP,Det),
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;
```

---

[6] Note that in the statements in definition (12), "voice", "subj", "arg1", etc do not denote constant DAGs but are labels. One should not confuse these two kinds of labels.

```
Det -> a,
Det.cat=det,
Det.det=a,
Det.number=sin;

Noun -> dog,
Noun.noun=dog,
sinnoun(Noun),
high_status_animal(Noun);
```

In DCG, an approximate equivalent might be:

```
np(N,Nbr,X,Det,Y,Z,Semconf) -->
      det(Det,Nbr1), noun(N,Nbr2,Semconf),
      {nbr_agr(np(N,Nbr,X,Det,Y,Z,Semconf),det(Det,Nbr1))} .

det(a,sin) --> [a].

noun(dog,Semconf) -->  [dog],
                       {sinnoun(noun(dog,Semconf))},
                       {high_status_animal(noun(dog,Semconf))}.
```

Here, as in RATP, it is possible to invoke predicates which constrain the structures being built. In DCG, these structures are terms with a fixed "arity", whereas in RATP they are DAGs which may refer *a priori* to an indefinite number of substructures.

In PATR, an approximate equivalent of these three rules would be:

```
NP -> Det Noun,
NP.cat=np,
Det.cat=det,
Noun.cat=noun,
NP.det=Det.det,
NP.number=Det.number,          % number_agreement(NP,Det)
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;

Det -> a,
Det.cat=det,
Det.det=a,
Det.number=sin;

Noun -> dog,
Noun.noun=dog,
Noun.cat=noun,                 % sinnoun(Noun)
```

```
Noun.number=sin,                          % high_status_animal(Noun)
X.semconf.animate=yes,                    % animate(X)
X.semconf.eatable=no;                     % noneatable(X)
```

The essential difference from RATP is that it is impossible in PATR to define classes of structures using predicates.

Furthermore, in cases where RATP can make unrestricted use of recursive predicates between them (cyclicity, see next section), it is generally impossible to translate an RATP grammar into PATR formalism.

Even when cyclicity is imposed, RATP's predicative mechanism allows for greater modularity in the definition of structures; it also makes it possible to account in a natural way for the disjunction of classes of structures (e.g., the predicate "voice treatment" in the grammar sample given).

2.3.2 RATP as a programming language

We have seen that predicate definitions in RATP have a structure very similar to that of predicate definitions in Prolog, with certain differences:

- DAGs instead of terms
- access paths

This is clearly not a matter of chance: predicate structure in RATP was directly inspired by that of Prolog, the main difference being due to the fact that the arguments of an RATP predicate are DAGs rather than terms (the consequences of this distinction are explored more fully in section 3).

In Prolog, a statement (that is, a "Horn clause") of the type:

$$p(T1,\ldots,Tm) \text{ <-- } q_1(T11, \ldots, T1m_1), \ldots, q_n(Tn1, \ldots, Tnm_n)$$

may be read as follows: if for every i (1(i(n) the m_i-uplet (T11, Tim_i) belongs to the relation q_i, then the m-uplet (T1,...,Tm) belongs to the relation p.

In other words, a Prolog program may be thought of as specifying ever more closely classes of n-uplets of terms, namely the classes associated with each of the program's predicates: one begins by slotting *ground-terms* into the classes corresponding to predicates appearing in "tail-less" (that is, unconditionally verified) statements, and as soon as a new term appears in the class q_i, it leads to the appearance of new terms in a class p. This process produces, in the limit, for each predicate in the program, a set of n-uplets of terms which satisfy it[7].

Thus a Prolog program can be seen as an effective means of describing and computing classes of structures, these structures being terms in this case.

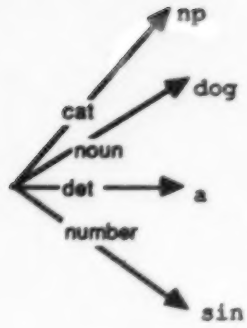
In RATP, the same mechanism is used: RATP predicate definitions make it possible to define classes of DAGs recursively. Thus, the predicate "sin_or_plu(X)" (definition B.3, p 13) produces a definition of the class of DAGs where "number" has the value of either "sin" or "plu"; the predicate "transitive_verb" (B.9) allows for the definition of all DAGs which are transitive(having "trans" as their transit (transitivity) value) verbs (that is, which have the category "v").

For instance, DAG A, which follows, belongs to the class associated with the predicate sin_or_plu[8]:

---

[7] The formal concept corresponding to this informal description is called **the least fixed point** in the program (Lloyd 1984).

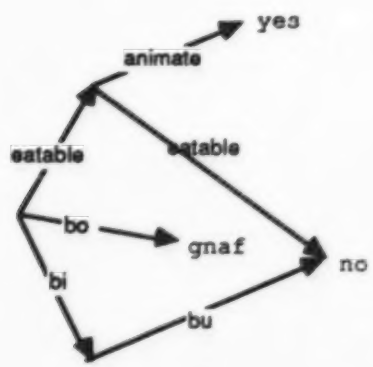
[8] Obviously, in the case of predicates with several arguments, n-uplets of DAGs are considered.



A

Predicates can "call" on each other; thus, for example, the predicate high status_animal (B.17) is defined with the assistance of the predicates "animate" and "noneatable".

DAG B below therefore belongs to the class "high_status_animal":



B

The "ever-closer" definition mechanism provides the language with great modularity and expressive power: as soon a class of structures has been defined, it can be used in turn to define new classes of structures.

Two basic types of cases can be distinguished, depending on the use made of this capacity.

First type: the expressive power of the language is limited by not allowing "cycles" in predicate definitions (cycle means that a predicate p1 can call on a predicate p2, which can in turn call on a predicate ..., which can call on predicate p1, whence the process began).

If this limitation is upheld, it can easily be shown that predicates can describe precisely those classes of structures which can be described solely with path equations, without recourse to predicates.

In other words, if this limitation is upheld, nothing is gained -- *from a theoretical standpoint* -- by allowing the use of predicates in the grammar:it would be sufficient to use grammar rules amplified by equations[9]; but a great deal is gained *practically* because of the modularity and the reuse capability introduced by predicates.

Second type: cycles are allowed in predicate definitions.

In this case, a predicate may call on itself, either directly or through other predicates. This is the situation in Prolog, and it is this option which makes Prolog a universal programming language, ie any class of "recursively enumerable" terms can be described by a predicate of this type.

Similarly, if cycles are allowed in predicate definitions in RATP, the language's descriptive power is considerably enhanced, as it is then possible to use RATP predicates to describe any class of recursively enumerable DAGs[10].

---

[9] This is the case in PATR II, as far as grammar is concerned, but PATR II introduces certain mechanisms which extend its scope, for example, in lexical matters.

[10] Strictly speaking, this is not accurate: in fact, it is possible to describe any recursively enumerable class of DAGs which has the property of being **free beyond a finite set of labels**, that is, such that

By way of example, it is easy to "program" addition and multiplication algorithms in RATP. A program of this type is shown in Appendix 2; this program defines a single-argument predicate mult(D) which is verified by a certain class of DAGs, those which represent, in a certain number coding convention, "multiplications".

For example, DAG A, which follows, represents a multiplication (three by two) and verifies the predicate mult(A).

---

the complement labels on a certain finite set of labels can have any value in the class. We shall not go into the details here, however.

```
arg1:
     car:
         one
     cdr:
         car:
             one
         cdr:
             car:
                 one
             cdr:
                 []
arg2:
     car:
         one
     cdr:
         car:
             one
         cdr:
             []
result:
     car:
         one
     cdr:
         car:
             one
         cdr:
             car:
                 one
             cdr:
                 car:
                     one
                 cdr:
                     car:
                         one
                     cdr:
                         car:
                             one
                         cdr:
                             []
```

In the same way as in Prolog, evaluation of an RATP predicate leads to either verification or enumeration of all DAGs[11] that verify the predicate,depending on the degree of instantiation of the DAG initially used as argument for the predicate. This evaluation is usually performed when the predicate is called on by a grammar rule, but may also occur independently of the grammar.

---

[11] that is to say, of course, <u>not</u> enumeration of <u>all</u> ground DAGs, but enumeration of a set of variable DAGs which collectively cover the set of all solutions.

3 - DAGS AND TERMS. QUESTIONS OF IMPLEMENTATION

3.1 DAGs and Terms

As we have seen in the foregoing section, RATP is an intermediate formalism between Unification Grammars (more specifically PATR II) and Logical Grammars (in particular DCG).

In addition, we have seen that the "programming language" aspect of RATP takes its chief theoretical inspiration from Prolog, the essential difference being due to the fact that the structures manipulated are DAGs and not terms.

The choice of Prolog as implementation language for RATP is therefore entirely natural.

One problem which must be solved in such implementation is that of representing DAGs in Prolog.

A priori, there are several possible choices -- for example, a DAG might be represented as a set of three-argument relations "arc(Node1,Node2,ArcLabel)" and unification of two DAGs achieved by manipulating these explicit graph representations --, but for reasons of effectiveness, it is by far preferable to use -- if possible -- the "built-in" term unification mechanisms of Prolog.

This assumes, however, that it is possible to encode DAGs as terms in such away that the unification of two DAGs amounts to a unification of terms[12].

---

[12] In this report, we do not describe the concept of unification of terms, for which the reader can consult Giannesini et al., 1986, for

3.2 Implementation of DAGs as Terms

From this standpoint, certain simplistic methods of encoding DAGs will not work; for example, one might want to encode a DAG as a list[13] of ordered pairs of the form (label_i, dag_i), where each dag_i is in turn a structure of the same type; one might then define the unification of two DAGs as a sort of "union" of two lists such that if a label appears in only one of the two lists, then the associated pair is "added" to the resulting list, and if the same label_i1=label_j2 appears in both lists l1 and l2, then the DAG associated with this label in the resulting list is the recursive unification of (lists representing) DAGs dag_i1 and dag_i2.

Unfortunately, as can easily be demonstrated, this method fails to satisfy a necessary condition for the unification of DAGs:

If at a given time (time t1) DAG A is unified with DAG B, and later (time t2) with C, and (time t3) B with D, then this implies that C and D are unifiable at time t4.

Two different methods may be proposed for solving this problem.

3.2.1 Encoding DAGs as incomplete lists

To solve the foregoing problem, Eisele and Doerre (Eisele et al., 1986) proposed an ingenious way of encoding DAGs.

---

example; nor do we treat the less well known concept of unification of DAGs, for which readers are referred to Shieber, 1986.

[13] that is, a term of a particular form.

In their approach, a DAG is encoded as an incomplete Prolog list L in the form:

L: C1.C2. ... .Cn.X,

where each Ci is a pair of the form c(Label,Value)

where, recursively, Value is an incomplete list (of the same form as L) representing a DAG.

The unification of two DAGs represented by L1 and L2 is then effected using the Prolog predicate

dag_unify(L1,L2)which is defined as follows:

```
dag_unify(L,L): !.
dag_unify(c(Label,V1).R1,L2):
      delete(c(Label,V2),L2,R2),
      dag_unify(V1,V2),
      dag_unify(R1,R2).

delete(C,C.L,L): ! .
delete(C,CC.L1,CC.L2): delete(C,L1,L2).
```

The predicate dag_unify has the remarkable property of possessing the following invariant:

after dag_unify has been successfully applied to L1 and L2, L1 and L2 have the form:

```
L1:   C11.C21.  ...  .Cn1.X
L2:   C12.C22.  ...  .Cn2.X
```

where C11.C21. ... .Cn1 and C12.C22. ... .Cn2 are a "permutation" of one another, where by permutation is meant: (a) that the lists of labels Et11 Et21 ... Etn1 (resp. Et12 Et22 ... Etn2), which are the initial elements of the pairs Ci1 (resp. Ci2), are permutations of one another, and (b) that the values Vi1 and Vj2 associated with the same label in L1 and L2 have themselves been

"dag-unified" and therefore recursively comply with the same invariant.

From this it follows that L1 and L2 represent the same DAG, AND that THEY RETAIN THIS PROPERTY THROUGHOUT THEIR SUBSEQUENT Prolog HISTORY, since X is common to both of them. Quite clearly, L1 and L2 are not unifiable AS Prolog terms.

This property guarantees that the unification of DAGs really have the requisite qualities. It does, however, have the following inconveniences:

- costly in terms of computation time (list scanning)
- creates a multiplicity of different Prolog terms (none of them unifiable as Prolog terms) all representing the same DAG.

This is why we have opted for the following method, which has the advantage of reducing the unification of DAGs to a simple Prolog unification of the terms which represent them.

3.2.2 Encoding DAGs as incomplete ternary trees

In order to be able to deal with DAG unification as a direct unification of Prolog terms, we have associated with each label used in the grammar a code in the form of a binary list (that is, a list of 0 or 1's) uniquely identifying this label[14].

Let us suppose that we wanted to encode, for example, the DAG

---

[14] The association of binary lists with labels is currently done "at compile time", but there is nothing to stop it being done "at run time" and adding new labels dynamically.

```
suj: Value1
obj: Value2
cat: Value3
```

(where Value1, Value2 and Value3 are themselves recursive encodings of DAGs not explicitly stated here).

Suppose also that the labels suj, obj and cat respectively are encoded as binary lists:

```
suj   0.1.1.nil
obj   1.nil
cat   nil
```

then the DAG shown above would be encoded as the term:

```
b(Value3,
  b(X1,
    X2,
    b(X3,
      X4,
      b(Value1,
        X5,
        X6)))
  b(Value2,X7,X8))
```

where X1...X8 are non-instantiated variables.

This structure can easily be understood by looking at the following informal description:

```
b(Value,          arrived!, i.e. the binary list is nil.

  Left,           the first element of the binary list is 0,
                  one "turns" left.

  Right,          le premier élément de la liste binaire est 0,
                  one "turns" right.
 )
```

This encoding guarantees that DAG unification will be reduced to direct unification (as understood in Prolog) of the terms representing them.

APPENDIX 1

A more complex RATP grammar

(translation into RATP of Shieber's (1986) grammar 3)

APPENDIX 1.1

The Grammar

```
S -> NP VP,
cat_s(S),
cat_np(NP),
cat_vp(VP),
S.head=VP.head,
S.head.form=finite,
VP.syncat.first=NP,
VP.syncat.rest=end;

VP -> V,
cat_vp(VP),
V.cat=v,
VP.head=V.head,
VP.syncat=V.syncat;

VP -> VP1 X,
cat_vp(VP),
cat_vp(VP1),
VP.head=VP1.head,
VP1.syncat.first=X,
VP1.syncat.rest=VP.syncat;

VP1 -> VP,
VP1=VP;

X -> VP,
X=VP;

X -> NP,
X=NP;

NP -> uther,
cat_np(NP),
ms(NP),
third(NP),
NP.head.trans=uther;

NP -> cornwall,
cat_np(NP),
ms(NP),
third(NP),
NP.head.trans=cornwall;

NP -> knights,
cat_np(NP),
mp(D),
third(NP),
NP.head.trans=knights;

V -> sleeps,
finite_3s(V),
```

```
sc_np(V),
V.head.trans.pred=sleep;

V -> sleep,
finite_sauf_3s(V),
sc_np(V),
V.head.trans.pred=sleep;

V -> sleep,
cat_v(V),
V.head.form=nonfinite,
sc_np(V),
V.head.trans.pred=sleep;

V -> storms,
finite_3s(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> stormed,
pastparticiple(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> storm,
nonfinite(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> has,
finite_3s(V),
have(V);

V -> have,
finite_sauf_3s(V),
have(V);

V -> persuades,
V.cat=v,
V.head.form=finite,
allow_type_control(V),
V.head.trans.pred=persuade;

V -> persuaded,
V.cat=v,
V.head.form=pastparticiple,
allow_type_control(V),
V.head.trans.pred=persuade;

V -> promises,
V.cat=v,
V.head.form=finite,
promise_type_control(V),
V.head.trans.pred=promise;

V -> promised,
V.cat=v,
V.head.form=pastparticiple,
```

```
promise_type_control(V),
V.head.trans.pred=promise;

V -> to,
V.cat=v,
V.head.form=infinitival,
V.syncat.first.cat=vp,
V.syncat.first.head.form=nonfinite,
V.syncat.first.syncat.rest=end,
V.syncat.first.syncat.first=V.syncat.rest.first,
V.syncat.rest.first.cat=np,
V.syncat.rest.rest=end,
V.head.trans.pred=V.syncat.first.head.trans;

cat_s(X):
X.cat=s;

cat_vp(X):
X.cat=vp;

cat_np(X):
X.cat=np;

cat_v(X):
X.cat=v;

ms(NP):
NP.head.agreement.gender=masculine,
NP.head.agreement.number=singular;

mp(NP):
NP.head.agreement.gender=masculine,
NP.head.agreement.number=plural;

third(NP):
NP.head.agreement.person=third;

nonfinite(V):
V.cat=v,
V.head.form=nonfinite;

finite_3s(V):
cat_v(V),
V.head.form=finite,
suj_of_v(V,SUJ),
SUJ.head.agreement.person=third,
SUJ.head.agreement.number=singular;

finite_sauf_3s(V):
cat_v(V),
V.head.form=finite,
suj_of_v(V,SUJ),
not_3s(SUJ);

not_3s(SUJ):
SUJ.head.agreement.person#third,
SUJ.head.agreement.number=singular;
```

```
not_3s(SUJ):
SUJ.head.agreement.person=third,
SUJ.head.agreement.number=singular;

suj_of_v(V,SUJ):
V.syncat=L,
V.syncat.rest=X,
freeze(X,suj_of_v_aux(V,SUJ,L));

suj_of_v_aux(V,SUJ,L):
L.rest=end,
L.first=SUJ;

suj_of_v_aux(V,SUJ,L):
L.rest=end,
L.rest=LL,
suj_of_v_aux(V,SUJ,LL);

pastparticiple(V):
V.cat=v,
V.head.form=pastparticiple;

pastpart(V):
V.head.form=pastparticiple;

have(V):
V.syncat.first.cat=vp,
V.syncat.first.head.form=pastparticiple,
V.syncat.first.syncat.rest=end,
V.syncat.first.syncat.first=V.syncat.rest.first,
V.syncat.rest.first.cat=np,
V.syncat.rest.first.head.agreement.person=third,
V.syncat.rest.first.head.agreement.number=singular,
V.syncat.rest.rest=end,
V.head.trans.pred=perfective,
V.head.trans.arg1=V.syncat.first.head.trans;

allow_type_control(V):
cat_v(V),
sc_np_np_tovp(V),
V.syncat.rest.first.syncat.first=V.syncat.first;

promise_type_control(V):
cat_v(V),
sc_np_np_tovp(V),
V.syncat.rest.first.syncat.first=V.syncat.rest.rest.first;


sc_np(V):
V.syncat.first.cat=np,
V.syncat.rest=end,
V.head.trans.arg1=V.syncat.first.head.trans;

sc_np_np(V):
V.syncat.first.cat=np,
V.syncat.rest.first.cat=np,
V.syncat.rest.rest=end,
```

```
V.head.trans.arg1=V.syncat.rest.first.head.trans,
V.head.trans.arg2=V.syncat.first.head.trans;

sc_np_np_tovp(V):
V.syncat.first.cat=np,
V.syncat.rest.first.cat=vp,
V.syncat.rest.first.head.form=infinitival,
V.syncat.rest.first.syncat.rest=end,
V.syncat.rest.rest.rest=end,
V.head.trans.arg1=V.syncat.rest.rest.first.head.trans,
V.head.trans.arg2=V.syncat.first.head.trans,
V.head.trans.arg3=V.syncat.rest.first.head.trans;
```

APPENDIX 1.2

Some sentences as analyzed by the grammar

```
52 /ul/dta/dymetman> ratpi

                        MPROLOG (2.1.0) LOGIC - LAB
                          (c) 1985 Logicware Inc.
                    PDSS Program Development Support System


                              RATP INTERPRETER

Which grammar file ? :
*s8

s8.ratp_output LOADED .

;;;;;;;; some analyses of the S "class"

: ?parse(s,uther.storms.cornwall.nil).

[uther,storms,cornwall]

 cat:
     s
 head:
     form:
         finite
     trans:
         arg1:
             uther
         arg2:
             cornwall
         pred:
             storm

Yes

: ?parse(s,uther.has.promised.knights.to.storm.cornwall.nil).

[uther,has,promised,knights,to,storm,cornwall]

 cat:
     s
 head:
     form:
         finite
     trans:
         arg1:
             arg1:
                 uther
```

```
            arg2:
                knights
            arg3:
                pred:
                    arg1:
                        uther
                    arg2:
                        cornwall
                    pred:
                        storm
            pred:
                promise
        pred:
            perfective

Yes

;;;;;It will be noticed that there is a difference between the verbs
"promise" and "persuade"; whereas the action is performed by arg 1 of
"promise", it is performed by arg 2 of "persuade".

: ?parse(s,uther.has.persuaded.knights.to.storm.cornwall.[]).

[uther,has,persuaded,knights,to,storm,cornwall]

 cat:
     s
 head:
     form:
         finite
     trans:
         arg1:
             arg1:
                 uther
             arg2:
                 knights
             arg3:
                 pred:
                     arg1:
                         knights
                     arg2:
                         cornwall
                     pred:
                         storm
             pred:
                 persuade
         pred:
             perfective

Yes
```

```
;;;;;Some "analyses" of the "class" NP. L is a variable here, so that
what really happens is a synthesis of all the np.

: ?parse(np,L).

[uther]

 cat:
     np
 head:
     agreement:
         gender:
             masculine
         number:
             singular
         person:
             third
     trans:
         uther


     L = [uther]
Continue (y/n) ?

[cornwall]

 cat:
     np
 head:
     agreement:
         gender:
             masculine
         number:
             singular
         person:
             third
     trans:
         cornwall


     L = [cornwall]
Continue (y/n) ?

[knights]

 cat:
     np
 head:
     agreement:
         person:
             third
     trans:
         knights


     L = [knights]
Continue (y/n) ?
```

```
NO

;;;;; Some "analyses" of the VP "class". L is a variable here, so that
what really happens is a synthesis o all the vp.

: ?parse(vp,L).

[sleeps]

 cat:
     vp
 head:
     form:
         finite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
             agreement:
                 number:
                     singular
                 person:
                     third
     rest:
         end


     L = [sleeps]
Continue (y/n) ?

[sleep]

 cat:
     vp
 head:
     form:
         finite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
             agreement:
                 number:
                     singular
     rest:
         end
```

```
    L = [sleep]
Continue (y/n) ?

[sleep]

 cat:
     vp
 head:
     form:
         finite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
             agreement:
                 person:
                     third
     rest:
         end


    L = [sleep]
Continue (y/n) ?

[sleep]

 cat:
     vp
 head:
     form:
         nonfinite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
     rest:
         end


    L = [sleep]
Continue (y/n) ?

[storms]

 cat:
     vp
 head:
     form:
         finite
     trans:
```

```
        pred:
            storm
    syncat:
        first:
            cat:
                np
            head:
        rest:
            first:
                cat:
                    np
                head:
                    agreement:
                        number:
                            singular
                        person:
                            third
            rest:
                end


    L = [storms]
Continue (y/n) ?  n
OK

;;;;;;;;;;;;;  An analysis of the VP "class".

: ?parse(vp,has.persuaded.knights.to.storm.cornwall.nil).

[has,persuaded,knights,to,storm,cornwall]

 cat:
    vp
 head:
    form:
        finite
    trans:
        arg1:
            arg2:
                knights
            arg3:
                pred:
                    arg1:
                        knights
                    arg2:
                        cornwall
                    pred:
                        storm
            pred:
                persuade
        pred:
            perfective
 syncat:
    first:
        cat:
            np
        head:
            agreement:
```

```
                number:
                    singular
                person:
                    third
        rest:
            end


Yes
: bye
*** The following module(s) have not been saved: ***
        unnamed_module
Do you want to exit (y/n) ?  y
Normal exit from MPROLOG PDSS
52 /ul/dta/dymetman> ^D
script done on Fri Mar 27 19:26:35 1987
```

APPENDIX 2

An RATP multiplier program

```
add(D):
D.arg1=nil,
D.arg2=D.result;

add(D):
D.arg1.car=one,
D.arg1.cdr=D1.arg1,
D.arg2=D1.arg2,
D.result.car=one,
D.result.cdr=D1.result,
add(D1);


mult(D):
D.arg1=nil,
D.result=nil;

mult(D):
D.arg1.car=one,
D.arg1.cdr=X,
D.arg2=Y,
D.result=Z,
D1.arg1=X,
D1.arg2=Y,
D1.result=Z1,
mult(D1),
D2.arg1=Y,
D2.arg2=Z1,
D2.result=Z,
```

```
add(D2);


mm(D):
D.arg2.car=one,
D.arg2.car=one,
D.arg2.cdr.car=one,
D.arg2.cdr.cdr=nil,
D.arg1.car=one,
D.arg1.cdr.car=one,
D.arg1.cdr.cdr.car=one,
D.arg1.cdr.cdr.cdr=nil,
mult(D);
```

BIBLIOGRAPHY

Bresnan, J. & Kaplan, R. 1983: "Lexical-Functional Grammar: a Formal System for Grammatical Representation", in J. Bresnan (ed.), The mental representation of grammatical relations, MIT Press.

Colmerauer, A. 1978: "Metamorphosis Grammars", in L. Bolc (ed.) Lecture Notes in Computer Science, Springer Verlag, vol. 63.

Dahl, V. 1985: "Logic Based Metagrammars for Natural Language Analysis", Technical Report, Computing Science, Simon Fraser University, Vancouver.

Eisele, A. & Dorre, J. 1986: "A Lexical-Functional System in Prolog", Proceedings of Coling 86.

Giannesini, F. Kanoui, H. Pasero, R. Van Caneghem, M. 1985, Prolog, Interéditions, Paris.

Kay, M. 1983, "Unification Grammar", Xerox Palo Alto Research Unit, Palo Alto.

Lloyd, J. 1984: Foundations of Logic Programming, Springer Verlag.

Pereira, F. 1981: "Extraposition Grammars", American Journal of Computational Linguistics, vol. 9:3.

Pereira, F. & Warren, D. 1980: "Definite Clause Grammars of Language Analysis", Artificial Intelligance, vol. 13.

Shieber, S. 1984: "The Design of a Computer Language for Linguistic Information", Proceedings of the ACL 1984.

Shieber, S. 1986: An Introduction to Unification-Based Grammar Formaliams, CSLI, Stanford.